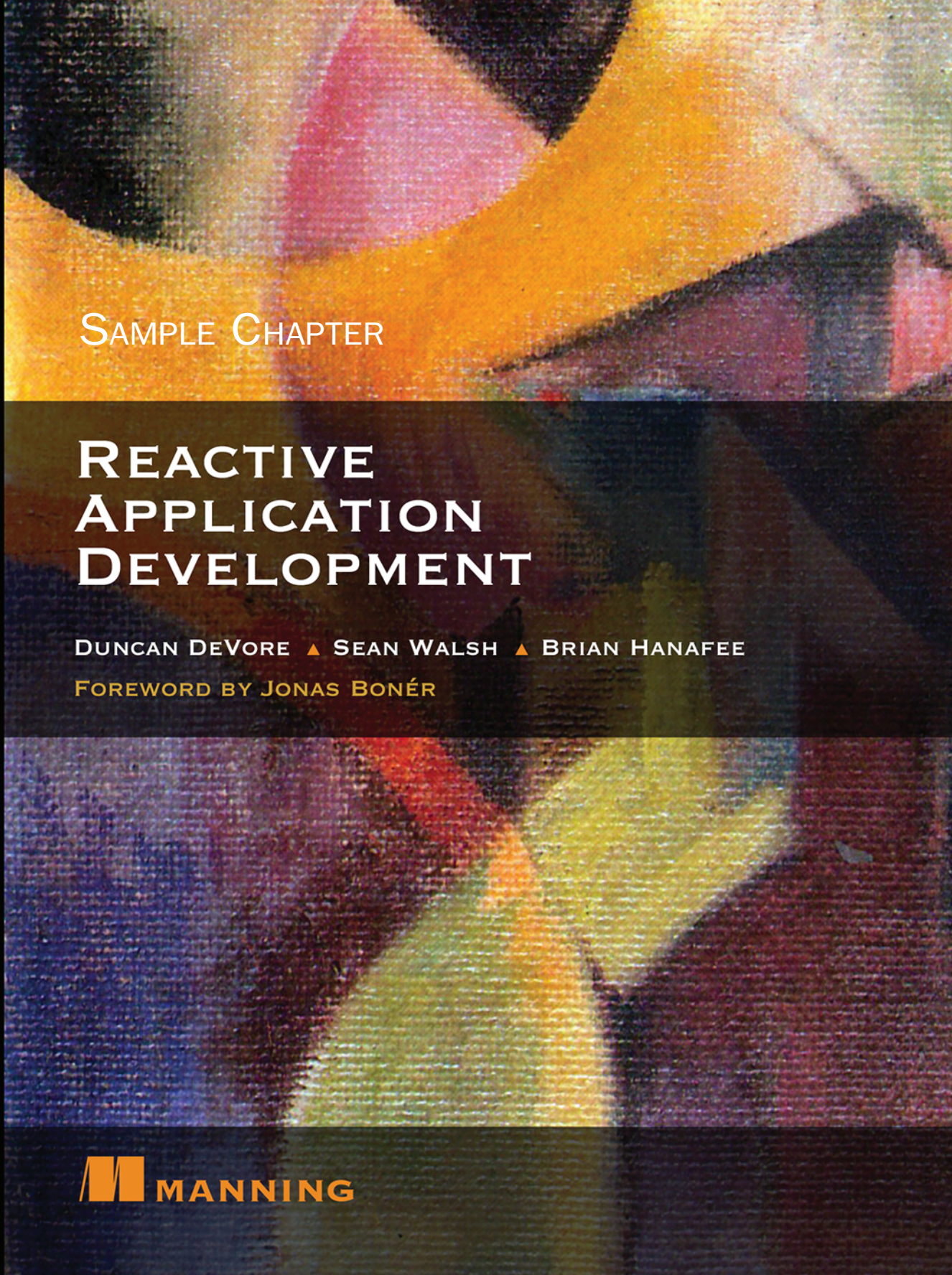
SAMPLE CHAPTER
REACTIVE APPLICATION DEVELOPMENT
DUNCAN DEVORE ▲ SEAN WALSH ▲ BRIAN HANAFEE
FOREWORD BY JONAS BONÉR
MANNING

Reactive Application Development

by Duncan DeVore, Sean Walsh and Brian Hanafee

Sample Chapter 7

brief contents

7 Reactive streaming

This chapter covers

- Seeing the dangers of unbounded buffers
- Safeguarding your application with backpressure
- Using Akka Streams in your application
- Integrating Akka Streams with other toolkits

In chapter 6, you learned how to cross actor system boundaries and send messages to remote actors. In this chapter, you learn how to prevent an application from being overwhelmed by too many messages. The reactive approach to regulating streams of messages so that they don't become floods is called *backpressure*.

Akka applies backpressure for you through Akka Streams. On the surface, Akka Streams is similar to other libraries you may have encountered, such as the `java.util.stream` package introduced in Java 8. You can take advantage of it directly by assembling stream sources, sinks, flows, and graphs without having to worry much about what's happening below the surface, as shown in section 7.3. Alternatively, you can use toolkits that are built on top of it, such as Akka HTTP (which we cover in chapter 9).

After learning about the Akka Streams library, you take it to the next level with the Reactive Streams application programming interface (API), which provides a standard interface to asynchronous streams and, as part of the specification, requires nonblocking backpressure. Reactive Streams is supported by many toolkits, including Akka, Java 9, and .NET. You can use it to integrate with other systems and apply backpressure across the connection.

We start by examining what happens when no backpressure exists.

7.1 Buffering too many messages

If you wanted to, you could configure millions of actors distributed across thousands of servers to send messages to a single actor running on your laptop. That single actor would expect to continue receiving messages one at a time, in keeping with the single-threaded contract it has with Akka. The result would be a huge backlog of unprocessed messages. Where would those messages be?

Ideally, the backlog of messages would be safe in the actor's mailbox. The default mailbox types are unbounded, so they can grow to accommodate a considerable number of unprocessed messages. Eventually, the application will run out of available memory to hold more messages, and the server may buffer additional data in system buffers. Usually, those buffers hold a small fraction of what an actor's mailbox can hold. When those buffers overflow too, the server is forced to start rejecting incoming messages, as shown in figure 7.1.

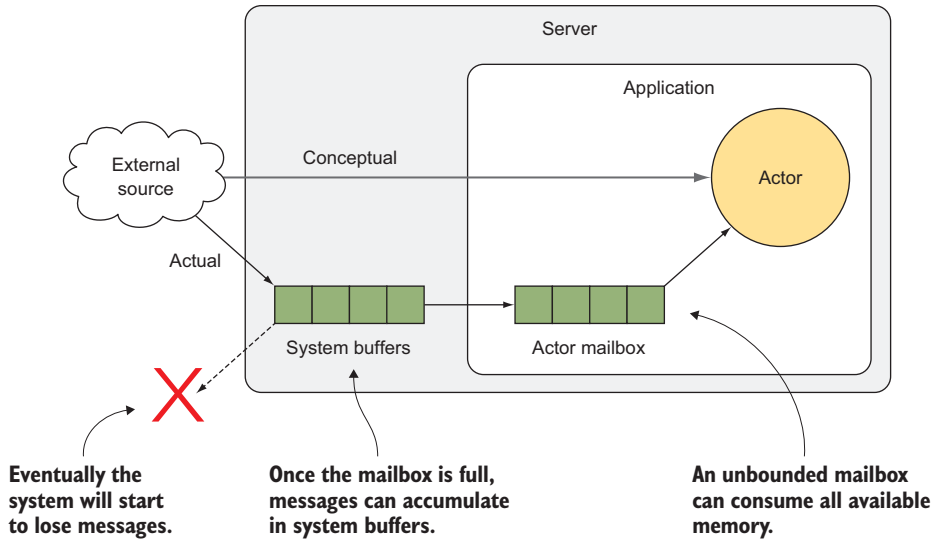

Figure 7.1 System buffers may attempt to handle messages that your application isn't ready to accept.

It may seem far-fetched to expect enough incoming data to overwhelm your system under normal load, especially if the system was designed with reactive principles and is capable of elastic growth. But consider some of the conditions that could lead to sudden increases in the numbers of messages that need to be processed:

- Your site appears in the news, and people all over the world start using it at the same time. This situation is sometimes called the "Reddit hug of death," because a link appearing on the front page of that site has been known to send surges of traffic to smaller sites that are ill-prepared to handle it.
- A system that your actor needs to complete processing is down for an extended period, so messages accumulate while recovery takes place.
- Other nodes in your application may fail or be unreachable, causing traffic to concentrate on a single node.
- Your application could be subject to a distributed denial-of-service attack. These attacks can exceed 1 terabit per second—enough to overwhelm any server they reach.

Your application may have to defend itself against unexpected increases in workload.

7.2 Defending with backpressure

At its root, the problem is that work is arriving faster than it's being processed. Eventually, no matter what you do, the application will be overwhelmed. The reactive solution is to slow the arrival rate of new work through backpressure.

Conceptually, the idea behind backpressure is simple. The data consumer tells the source how much data it's prepared to accept, and the source sends no more than that amount. You might object that backpressure moves the problem from one system to another, and you'd be correct. The beauty of backpressure, however, is it can keep going: each component can push back on the one before it, going all the way back to the original source if necessary, which turns out to be highly effective in real systems.

7.2.1 Stopping and waiting

Requiring the publisher to wait for a signal before sending each message provides a primitive form of backpressure. Figure 7.2 illustrates this approach in a publish/subscribe system with a single subscriber. The subscriber provides the backpressure by requiring the publisher to wait for an OK message before sending a message containing work for the subscriber to process, preventing a queue of messages between publisher and subscriber from accumulating. Instead, the publisher has to hold each message until it knows that the subscriber is ready to process it. Although holding each message may appear at first glance to be equivalent to the publisher's making synchronous calls to the subscriber, it is different. Publisher and subscriber are sending asynchronous messages to each other, so the message exchange is nonblocking.

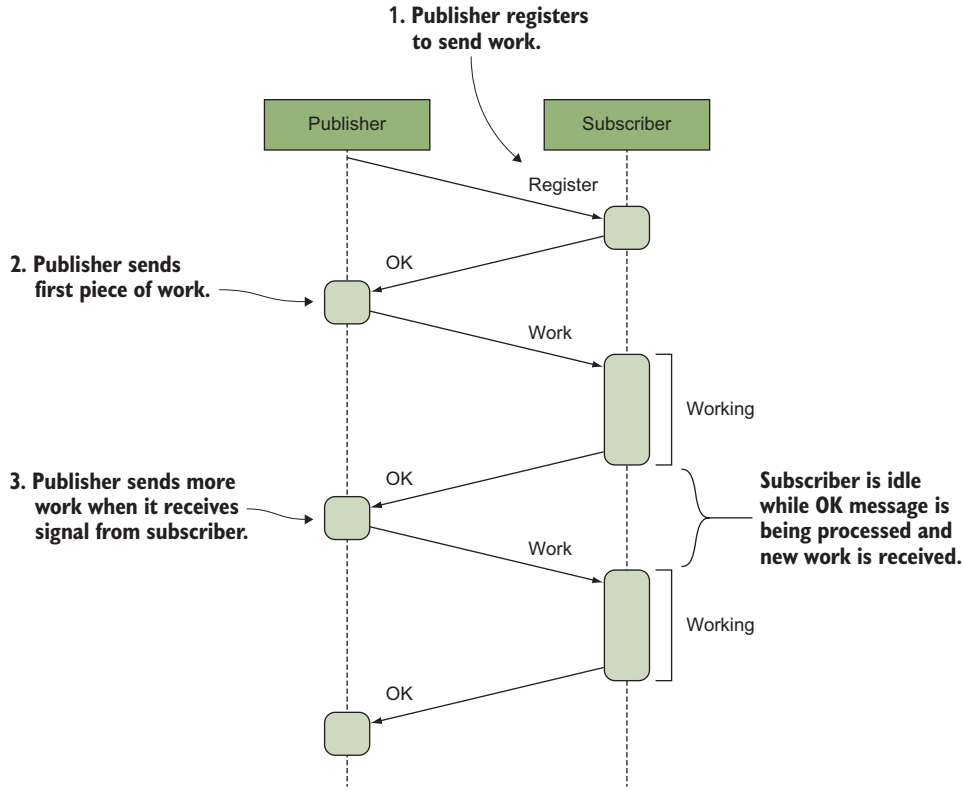

Figure 7.2 Positive acknowledgement per message idles available resources.

A minimal implementation of backpressure with positive acknowledgement could consist of a `Publisher` actor and a `Subscriber` actor. The `Publisher` waits for an OK message before sending a Work message to the `Subscriber`, as follows.

Listing 7.1 Publisher actor with OK processing

```
import akka.actor.{Actor, ActorRef}
import Subscriber.{Register, Work}

object Publisher {
  case object Ok
}

class Publisher(subscriber: ActorRef) extends Actor {
  override def preStart =
    subscriber ! Register
  override def receive = {
    case Publisher.Ok =>
      subscriber ! Work("Do something!")
  }
}
```

Message to tell the publisher it's OK to send work

Sends an initial message to start the process

Publisher sends work when it receives an OK message.

When the subscriber completes a piece of work, it replies with an OK message signaling that it's ready to accept another, as shown in the following listing. After responding with the OK message, the subscriber waits idly for another piece of work.

Listing 7.2 Subscriber with OK response

```
import akka.actor.Actor

object Subscriber {
  case object Register
  case class Work(m: String)
}

import Subscriber.{Register, Work}
class Subscriber extends Actor {
  override def receive = {
    case Register =>
      sender() ! Publisher.Ok
    case Work(m) =>
      System.out.println(s"Working on $m")
      sender() ! Publisher.Ok
  }
}
```

OK request to send initial work

Performs the requested work

Tells the publisher it's OK to send more work

The driver, shown in the following listing, sets up the actor system and the two actors as you'd expect: it configures the actor system, waits a few seconds for it to do some work, and shuts down. You can download the example from http://mng.bz/71O3 and run it by using the command `sbt run`.

Listing 7.3 Driver to start and stop the example

```
import akka.actor.{ActorRef, ActorSystem, Props}

object Main extends App {
  val system: ActorSystem = ActorSystem("StopWait")

  val subscriberProps = Props[Subscriber]
  val subscriber: ActorRef = system.actorOf(subscriberProps)

  val publisherProps =
  ➥ Props(classOf[Publisher], subscriber)
  val publisher: ActorRef = system.actorOf(publisherProps)

  Thread.sleep(10000)
  system.terminate()
}
```

The publisher gets a reference to the subscriber.

Waits a few seconds and then shuts down the whole actor system

Congratulations—you've implemented a stream protocol with backpressure. At this stage, the example is rudimentary. You see a lot of messages scrolling when you run it, but it's quite inefficient. After completing each piece of work, the subscriber has to wait for an OK message to get back to the requestor and then wait for the message

containing the next piece of work to arrive before it starts working again. Another problem evident in the driver program is that it lacks a clean shutdown, which can lead to the loss of an unfulfilled work request. You may encounter a few warnings about dead letters.

In the next two sections, we explore how to make the stream implementation slightly less rudimentary. First, we consider how to keep the subscriber busy.

7.2.2 Signaling for more than one message

You can reduce the subscriber's idle time by telling the publisher that it's OK to send multiple requests rather than one. The number of requests allowed can be fixed or passed as a parameter in the message. The publisher keeps a running count of the number of messages that it's allowed by the subscriber to send. The subscriber is responsible for deciding when it can allocate more messages to a publisher. In figure 7.3, each message from the subscriber back to the publisher says that it's OK to send three more work requests.

The subscriber doesn't have to wait until it receives three work requests to tell the publisher that it can send more. Requesting additional work before the subscriber fully completes processing the current requests is one way to ensure that the sub-

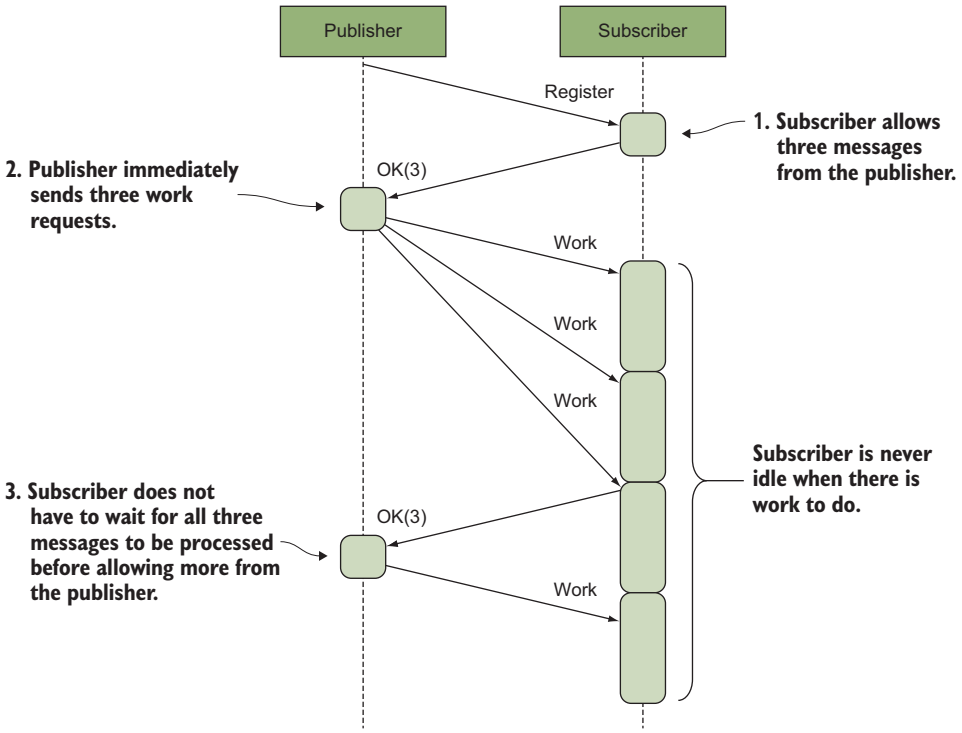

Figure 7.3 The subscriber tells the publisher how many messages it's prepared to accept.

scriber has a continuous supply of work. The additional messages sit in an inbox queue until the subscriber processes them, and the subscriber is responsible for ensuring that it doesn't request more messages than it can handle.

Similarly, when the subscriber sends a message to the publisher to inform it that three messages have been allocated for it to send, the publisher is *not* obligated to send that many messages. The number allowed is a maximum. If the publisher doesn't have that many messages left, it can end the stream.

At this point, your stream implementation can apply enough backpressure to keep the subscriber busy without being overwhelmed, but it still expects to run forever.

> **Signaling for multiple messages isn't microbatching**
>
> A signal that a stream is ready to receive multiple messages means that the sender is allowed to send a set number of *individual* messages. The messages are still received one at a time. This process is different from a technique called microbatching, which is commonly used to optimize big data systems. With microbatching, the system accumulates messages that are ready for processing until some limit is reached. Usually, the limit is a fixed number of messages or some maximum time between batches. When it reaches the limit, the system passes all the accumulated messages at the same time. Then the processor has to handle the microbatch as a unit, mixing infrastructure considerations with business logic.

7.2.3 Controlling the stream

Either the publisher or the subscriber may end the stream. Because the publisher and subscriber are asynchronous, the details of ending the stream are a little bit different, depending on which actor ends it. If the publisher is ending the stream, it has to stop sending messages. If the subscriber wants to end the stream, it sends a message back to the publisher. That message may take time to arrive and be processed by the publisher. In the meantime, messages that were sent before the cancellation was processed continue to arrive at the subscriber, as shown in figure 7.4. It's up to the subscriber to decide what to do with messages that arrive after it sent a cancellation message: ignore them or process them. The publisher has no way to know whether the messages arrived at the subscriber.

Whether the publisher or subscriber ends the stream, a completion message usually is sent at the end of the steam. This message informs the subscriber so that it can release resources and perform any final processing. The completion message may take the form of a success message (if the stream is terminating normally) or a failure message. When the completion message arrives, no further messages should follow.

When you use Akka Streams, you don't need to write code for all the intricacies of managing the request count and flow control, so we won't extend the example in this section. We return to it in section 7.4, however, when we discuss the Reactive Streams API.

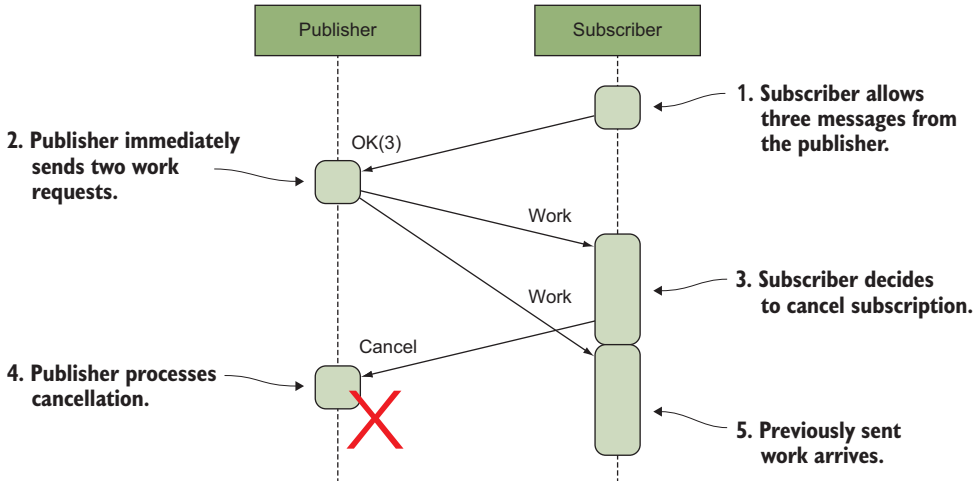

Figure 7.4 Some messages may continue to arrive after a subscription is canceled.

7.3 Streaming with Akka

Akka Streams are graphs assembled from processing stages. Each stage can exert backpressure on earlier stages. A basic stream consists of a *source* and a *sink*, which are combined to create a *flow* that you can execute. A complete stream could be as simple as

```
source.to(sink).run()
```

To see how to use Akka Streams, give the RareBooks librarians from chapter 6 a new job: loading entries to the catalog. The stream consists of a source, some intermediate flows, and a sink. The source is a stream of bytes read from a comma-separated file. Then the flows convert the stream of bytes (a continuous `ByteString`) into a stream of `BookCard` entries:

1. A *framing* flow uses the line separators that it encounters in the stream to produce a stream consisting of a separate `ByteString` for each line.
2. A *mapping* flow parses each comma-separated `ByteString` to produce a stream of arrays of `Strings`.
3. Another *mapping* flow converts each array of `Strings` to a `BookCard`, producing a stream of `BookCard` entries.

The stream of `BookCard` entries terminates with a sink. The sink sends the `BookCard` as a message to a `librarian` actor, which adds the card to the catalog. You may recall from chapter 6 that the `librarian` actor takes time to perform each task. You don't have to worry about flow control. Behind the scenes, Akka Streams applies backpressure so that `BookCard` entries don't come too fast.

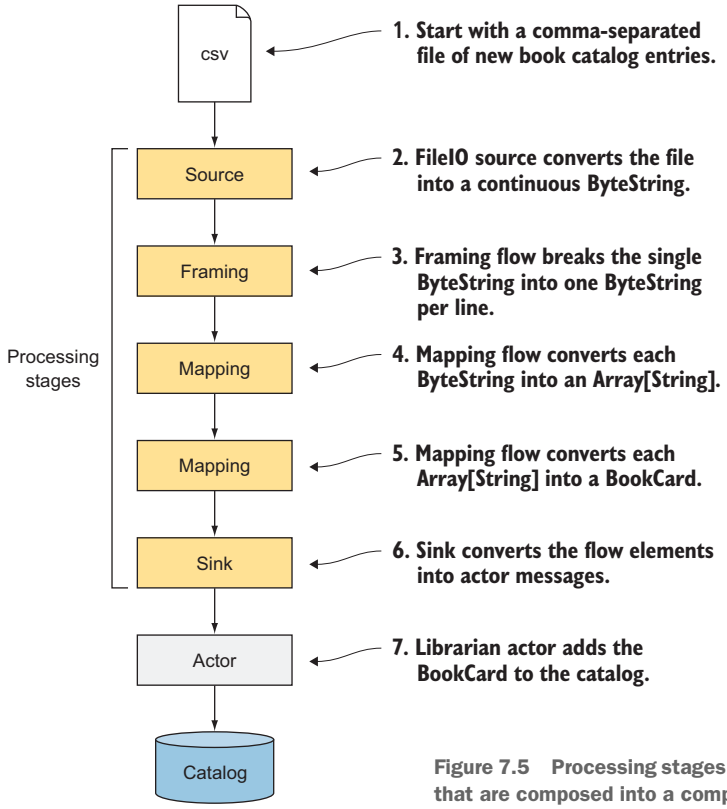

Figure 7.5 Processing stages should be simple operations that are composed into a complete processing graph.

Figure 7.5 shows the components you need to create. You can download the source code for this chapter from http://mng.bz/71O3.

7.3.1 Adding streams to a project

The catalog loader is a stand-alone application with its own `build.sbt`. Akka Streams is separate from the Akka core, so start by adding both modules as dependencies, as follows.

Listing 7.4 build.sbt for the catalog loader

```
val akkaVersion = "2.5.4"

scalaVersion := "2.12.3"

name := "catalogLoader"

libraryDependencies ++= Seq(
  "com.typesafe.akka"  %% "akka-actor"              % akkaVersion,
  "com.typesafe.akka"  %% "akka-stream"             % akkaVersion
)
```

The complete catalog loader has many moving parts, so you'll build it in phases.

> **TIP** The Alpakka initiative (https://github.com/akka/alpakka) has connectors that handle integration with Amazon Web Services (AWS), Google Cloud, Microsoft Azure, and several queueing packages. The project also maintains a list of externally developed connectors.

7.3.2 Creating a stream source from a file

The goal of the first phase (listing 7.5) is to read a file and print the contents. The source is based on a file path, and the sink is a `println()`. Not surprisingly, running the stream requires an actor system. The actor system is no different from the actor systems for RareBooks introduced in chapter 6. The stream also requires an `ActorMaterializer`, which is new. The job of the materializer is to transform the flow into processors to be executed by actors.

Listing 7.5 Stream to read a file

```
package com.rarebooks.library

import java.nio.file.Paths

import akka.actor.ActorSystem
import akka.stream.ActorMaterializer
import akka.stream.scaladsl._

import scala.concurrent.Await
import scala.concurrent.duration.Duration

object Cataloging extends App {
  implicit val system =
  ➥ ActorSystem("catalog-loader")
  implicit val materializer = ActorMaterializer()

  val file = Paths.get("books.csv")

  val result = FileIO.fromPath(file)
      .to(Sink.foreach(println(_)))
      .run()

  Await.ready(result, Duration.Inf)
  system.terminate()
}
```

Processing stages execute in this system.

Transforms the stages into processors

Be sure that this file is in the current directory.

Stream source based on the file

Attaches the sink

Starts the stream processing

Waits for the stream to complete

Shuts down the actor system

Use `sbt run` to execute the stream. If everything is working, `sbt run` outputs a very long line like this:

```
ByteString(…)
```

The reason is that Akka `FileIO` produces an `akka.util.ByteString`, which is an optimized data type for working with streams of raw bytes.

7.3.3 Transforming the stream

The next phase of development is transforming those raw bytes into a stream of `BookCard`s that the librarian can add to the catalog. The transformation comprises the three flows described in the preceding sections: converting the continuous-source `ByteString` to single-line `ByteString`s, parsing each line into an array of `String`s, and converting each array to a `BookCard`.

Decoding a continuous stream into a stream of discrete elements is called *framing*. The Scala domain-specific language (DSL) has a function to generate a flow that frames lines around a delimiter, which in this case is a newline. The input is a single `ByteString`, and the output is a stream of individual `ByteString`s.

Converting each `ByteString` to an array of `String`s is a mapping constructed from a couple of utility functions. Turning that mapping into a flow is easy because `Flow` has a `map` function for that purpose. Following is the resulting application.

Listing 7.6 Transformation of a file into a stream of BookCards

```
// ... previous imports
//
import akka.util.ByteString
import LibraryProtocol.BookCard

object Cataloging extends App {
  implicit val system =
  ➥ ActorSystem("catalog-loader")
  implicit val materializer = ActorMaterializer()

  val file = Paths.get("books.csv")

  private val framing: Flow[ByteString, ByteString, NotUsed] =
    Framing.delimiter(ByteString("\n"),
      maximumFrameLength = 256,
      allowTruncation = true)

  private val parsing: ByteString => Array[String] =
    _.utf8String.split(",")

  private val conversion: Array[String] => BookCard =
    s => BookCard(
      isbn = s(0),
      author = s(1),
      // ... remaining fields
    )

  val result = FileIO.fromPath(file)
    .via(framing)
    .map(parsing)
    .map(conversion)
    .to(Sink.foreach(println(_)))
    .run()
  Await.ready(result, Duration.Inf)
  system.terminate()
}
```

Contains the utf8String conversion used in parsing

Declares the framing function

Declares the parsing function

Declares the conversion function

Frames the ByteString by line

Parses each line into an Array[String]

Converts to BookCard

Use `sbt run` to execute the stream. This time, you see individual BookCards.

Now take a step back to look at what you've accomplished. You've created five processing stages: a source, three flows, and a sink. Together, they read a file and transform it into a stream of `BookCard` entries for the RareBooks catalog.

> **TIP** It's worth spending some time to examining the scaladoc for `Flow`. It has a rich set of functions similar to those in the Scala collections library, including basics such as `filter`, `map`, `fold`, and `reduce`.

7.3.4 Converting the stream to actor messages

The next phase is replacing the sink that generates `println()` messages with one that sends messages to the `librarian` actor. The application has a reference to the actor system, so one approach is to use `tell`, as in the following.

Listing 7.7 Sink sending messages directly to an actor

```
// ...
val librarian: ActorRef
// ...
val result = FileIO.fromPath(file)
  .via(framing)
  .map(parsing)
  .map(conversion)
  .to(Sink.foreach(card => librarian ! card))
  .run()
```

Use tell to send the card directly to the Librarian actor.

For a somewhat richer interface, you can use `Sink.actorRef` to send the messages automatically. That function adds a final message that's sent to the actor when the stream is complete. Whether you choose `tell` or `Sink.actorRef`, each message to the actor behaves in the usual way. That is, each message is nonblocking, asynchronous, and one-way, which in this case presents a problem: no backpressure!

To apply backpressure from the actor to the stream, you need to use the more complex `Sink.actorRefWithAck` function. This function takes several parameters:

- *`ref`*—A reference to the actor.
- *`onInitMessage`*—A message sent before any elements from the stream.
- *`ackMessage`*—A message returned from the actor to acknowledge each request. The sink must receive this message after it sends `onInitMessage` and before it sends any stream elements, and the sink also must receive this message after each stream element.
- *`onCompleteMessage`*—A message sent to the actor when the stream completes successfully.
- *`onFailureMessage`*—A message sent to the actor if the stream completes with failure.

As you can see, the `Librarian` actor has to handle a few new messages. First, add those messages to the `LibraryProtocol`, as follows.

Listing 7.8 The `LibraryProtocol` extended to interact with the stream

```
case object LibInit
case object LibAck
case object LibComplete
case class LibError (t: Throwable)
```

To make the extended protocol work, you first need to make a few changes in the Librarian's receive function so that it's prepared to accept the new BookCards and respond as follows.

Listing 7.9 `Librarian` actor extended to add catalog entries

```
// ... start with the familiar Librarian you used in chapters 3, 4, and 6
  private def ready: Receive = {
    // ... preexisting match cases elided
    case LibInit =>
      log.info("Starting load")
      sender() ! LibAck
    case b: BookCard =>
      log.info(s"Received card $b")
      Catalog.books = Catalog.books + ((b.isbn, b))
      sender() ! LibAck
    case LibError(e) =>
      log.error("Load error", e)
    case LibComplete =>
      log.info("Complete load")
```

Allows the stream to start sending new books

Adds the new book to the catalog

Allows the stream to send another book

No special processing is needed for completion or error.

Now you've done all the preparatory work to complete the final phase of the example by adding `actorRefWithAck` to the processing stream.

7.3.5 *Putting it together*

Following is the complete processing stream.

Listing 7.10 `Sink.actorRefWithAck` to exert backpressure from the actor

```
// ...
val librarian: ActorRef
import LibraryProtocol._
// ...
val result = FileIO.fromPath(file)
  .via(framing)
  .map(parsing)
  .map(conversion)
  .to(Sink.actorRefWithAck(
    librarian, LibInit, LibAck, LibComplete, LibError)
  .run()
```

Imports the messages to interact with the librarian

Sink with the backpressure messages defined

When you run this application, it streams the file of books into new entries in the catalog. Now is a great time for you to experiment with backpressure! Here are a few things to try:

- Have the librarian take more time adding a card entry. Instead of sending a `LibAck` message right away, use the technique you learned in chapter 4 to schedule the acknowledgement message after a short delay.
- The librarian takes time researching requests from customers. Start up a few instances of the customer application and send some requests while the librarian is busy adding new card entries.

As you explore Akka Streams, you'll discover that they can go beyond simple linear flows that consist of a source, flows, and a sink. Streams can be assembled into graphs that have multiple inputs or outputs at each stage. Table 7.1 defines some terms you may encounter that describe different stream processing stages.

Table 7.1 Akka Streams processing stages may be categorized based on the number of inputs and outputs.

Type	Inputs	Outputs
Source	*	1
Sink	1	*
Flow	1	1
Fan-In	multiple	1
Fan-Out	1	multiple
BidiFlow	multiple	multiple

* External to the stream, such as a file connector

7.4 Introducing Reactive Streams

So far in this chapter, you've learned about backpressure as a reactive technique and applied it to Akka's Streams library. Akka Streams is built on backpressure with actor-based underpinnings. Other streams could be built on different frameworks but still support backpressure. *Reactive Streams* is "an initiative to provide a standard for asynchronous stream processing with non-blocking back pressure" (from www.reactive-streams.org). In other words, it's a distillation of the core features that a reactive implementation needs to provide. Reactive Streams provides a common language that allows different reactive implementations to interoperate.

Akka isn't the only implementation. Most notably, Reactive Streams is incorporated into Java 9 through Java Enhancement Proposal JEP-266. Spring Framework version 5 incorporates Reactive Streams through Project Reactor (https://projectreactor.io). Other implementations include RxJava, which is part of the ReactiveX project (https://reactivex.io), Ratpack (https://ratpack.io), and Eclipse Vert.x (http://vertx.io). More

implementations are appearing regularly. Reactive Streams includes a Technology Compatibility Kit (TCK) to help validate new implementations as they appear.

Reactive Streams should be viewed as APIs for providers. If you've settled on using a single toolkit such as Akka for your entire application, you don't use it directly. If you want to get two reactive systems to interoperate, and those systems don't already have a connector, consider using Reactive Streams to connect them.

The entire Reactive Streams API consists of four small interfaces:

- *Publisher*—The provider of a stream of elements
- *Subscriber*—The consumer of a stream of elements
- *Subscription*—The interface used for a subscriber to signal a publisher
- *Processor*—A processing stage that obeys the contracts of both publisher and subscriber

NOTE Reactive Streams doesn't have any dependencies on Akka. Implementations don't have to be based on the actor model at all.

7.4.1 Creating a Reactive Stream

Creating a Reactive Stream between two providers is easy. All you need is a reference to a `Publisher` from one provider and a `Subscriber` from the other. The heavy lifting is performed by the provider's implementations of the interface (figure 7.6):

1. The application calls the `subscribe` method on the `Publisher`, passing a reference to the `Subscriber`.
2. The `Publisher` creates a `Subscription`.
3. The `Publisher` calls the `onSubscribe` method on the `Subscriber`, passing a reference to the newly created `Subscription`.

At this point, the `Subscriber` uses the `Subscription` to start sending asynchronous signals back to the `Publisher`.

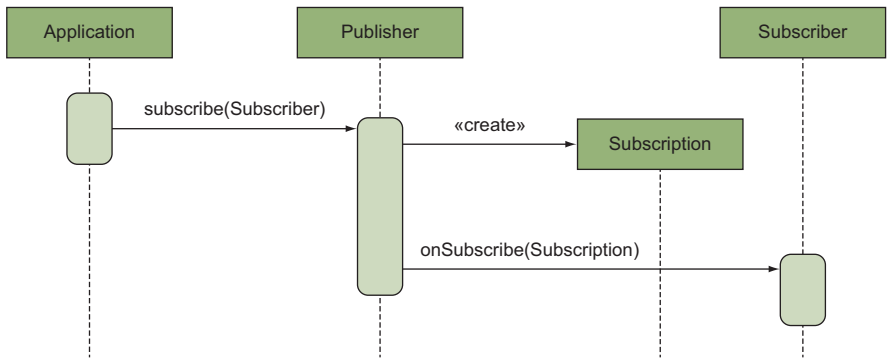

Figure 7.6 The application initializes a reactive stream by calling the publisher with a reference to the subscriber. The publisher creates a subscription and passes it to the subscriber.

WARNING Reactive Streams doesn't allow the same `Subscriber` to have multiple `Subscriptions` to the same `Publisher`. Calling `Publisher.subscribe` more than once for the same `Subscriber` may produce exceptions or unpredictable behavior.

7.4.2 Consuming the Reactive Stream

The `Subscriber` has two signals that it can send back to the `Publisher`; it can request messages or cancel the subscription. No messages flow until the subscriber starts the flow with the first call to request messages.

The `Publisher` keeps calling the `Subscriber`'s `onNext` method with new messages until it has sent as many as requested. If the `Publisher` runs out of messages or if an error occurs, the `Publisher` signals the `Subscriber` by using `onComplete` or `onError`, respectively (figure 7.7).

The `Subscriber` can request that the `Publisher` stop sending messages by canceling the `Subscription`. In that event, the publisher is required to stop sending messages eventually, but it may not stop immediately.

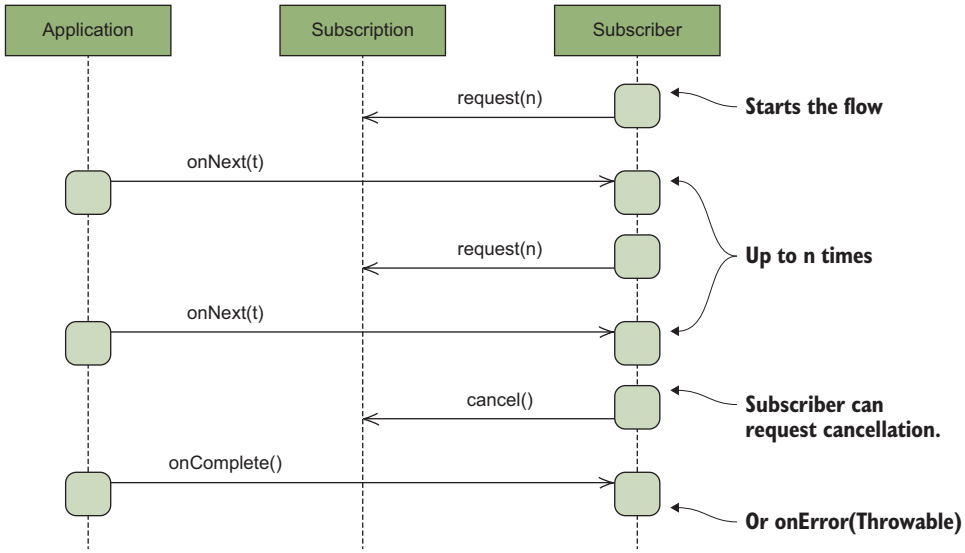

Figure 7.7 The publisher can end a stream by calling onComplete or onError. The subscriber can asynchronously request the publisher to end the stream by calling cancel.

TIP The default mailbox in Akka is unbounded, but several of the other implementations are bounded. If you're integrating with a Reactive Streams implementation, be sure not to request more messages than your application's inbox can hold. Otherwise, you could be subject to the overflowing-buffer conditions discussed at the beginning of this chapter.

7.4.3 Building applications with streaming

It seems that the software world is heading to streaming. You find streams in Apache Spark, Storm, Samza, Apex, Flink, Kafka, and more. If you stop and reflect for a moment, you realize that the move to streaming is a consequence of software that better reflects the real world, because the real world has a lot of asynchronous events. No matter how many incoming messages a system can handle at the same time, a few too many events might arrive and spoil the fun. In the real world, the solution is to slow the influx to something that the system can handle. Reactive streaming brings the same solution to software by telling the source to slow down so the application can catch up. This chapter showed you how to do that. In the next chapter, you put this knowledge to use with two types of messages that often arrive in a stream: commands and events.

Summary

- Bursts of message traffic can overflow into unexpected parts of your system. Overflowing buffers damage resiliency because the system relies on lower levels to recover.
- Backpressure defends your application by telling clients how many messages it's prepared to handle. Backpressure is implemented according to the same reactive principles as the rest of the system, so it's nonblocking, message-driven, and asynchronous.
- Akka Streams are constructed from processing stages. A typical stream consists of a source, some flows, and a sink. The Scala DSL is a rich library for assembling commonly needed processing stages.
- A stream of events can be converted to messages that can be processed by an actor.
- Reactive Streams is an API for implementing backpressure. You can use it to identify messaging frameworks with backpressure support and to create more portable reactive applications. Implementations include Akka Streams, Java 9, Spring 5, and .Net.

SOFTWARE DEVELOPMENT/JAVA

Reactive Application Development

DeVore ▲ Walsh ▲ Hanafee

Mission-critical applications have to respond instantly to changes in load, recover gracefully from failure, and satisfy exacting requirements for performance, cost, and reliability. That's no small task! Reactive designs make it easier to meet these demands through modular, message-driven architecture, innovative tooling, and cloud-based infrastructure.

Reactive Application Development teaches you how to build reliable enterprise applications using reactive design patterns. This hands-on guide begins by exposing you to the reactive mental model, along with a survey of core technologies like the Akka actors framework. Then, you'll build a proof-of-concept system in Scala, and learn to use patterns like CQRS and Event Sourcing. You'll master the principles of reactive design as you implement elasticity and resilience, integrate with traditional architectures, and learn powerful testing techniques.

What's Inside

- Designing elastic domain models
- Building fault-tolerant systems
- Efficiently handling large data volumes
- Examples can be built in Scala or Java

Written for Java or Scala programmers familiar with distributed application designs.

Duncan DeVore, **Sean Walsh**, and **Brian Hanafee** are seasoned architects with experience building and deploying reactive systems in production.

"Packed with hard-won wisdom and practical advice that will set you on the path toward effective reactive application development."

—From the Foreword by Jonas Bonér
Creator of Akka

"The ultimate reference on reactive application development, with Scala code using Akka as a bonus!"

—Jean-François Morin, Laval University

"Explains complex concurrency problems in simple words with lots of realistic examples."

—Shabeesh Balan
Prime Focus Technologies

"Very well written and reflects the authors' expertise ... teaches you how to build modern distributed applications using reactive design patterns."

—Subhasis Ghosh, Alliant Credit Union

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/reactive-application-development

ISBN-13: 978-1-61729-246-0
ISBN-10: 1-61729-246-X
54999
9 781617 292460

MANNING

US $ 49.99 | Can $ 65.99